

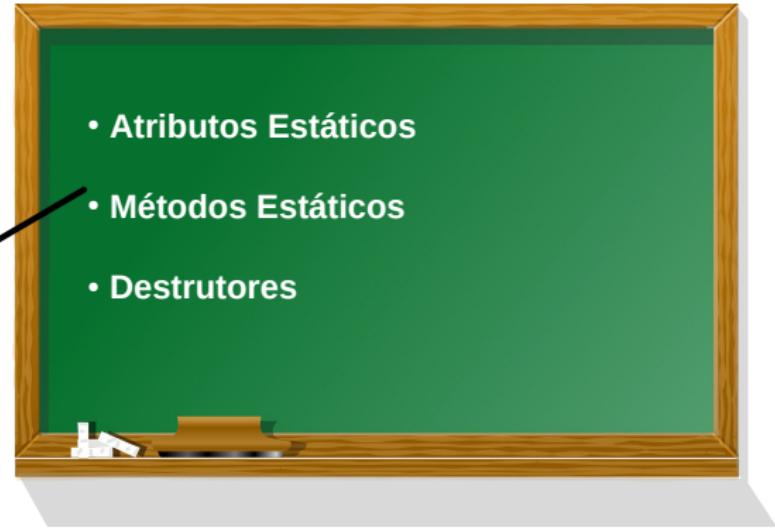
# Atributos Estáticos, Métodos Estáticos e Destruidores

Luiz Henrique de Campos Merschmann  
Departamento de Computação Aplicada  
Universidade Federal de Lavras

[luiz.hcm@ufla.br](mailto:luiz.hcm@ufla.br)



# Na Aula de Hoje



# Atributos Estáticos

- ▶ Suponha que você está desenvolvendo um sistema para uma agência de uma instituição bancária.
- ▶ A estratégia de venda dos produtos (consórcio, seguros, planos de previdência etc.) nessa agência depende do número de contas abertas na mesma.
- ▶ Você definiu a classe **ContaBancaria** para tratar os comportamentos de cada uma das contas abertas.
- ▶ Precisamos então armazenar em algum local a quantidade de contas abertas.
- ▶ Qual a melhor forma de guardar isso?
  - ▶ Na própria classe?
  - ▶ Em outra classe?
  - ▶ Como?



# Atributos Estáticos

- Se colocarmos essa informação em uma classe que não seja a **ContaBancaria**, teríamos que lembrar de incrementar a variável *totalContas* toda vez que um novo objeto da classe **ContaBancaria** fosse criado.

```
1 /** Classe principal que nos permite utilizar
2 * objetos da classe ContaBancaria
3 */
4 public class ContaBancariaTeste
5 {
6     public static void main(String[] args)
7     {
8         int totalContas = 0;
9
10        ContaBancaria conta1 = new ContaBancaria(1234,"Luiz Henrique");
11        totalContas = totalContas + 1;
12
13        ContaBancaria conta2 = new ContaBancaria(5678,"Marco Aurélio");
14        totalContas = totalContas + 1;
15    }
16 }
```



Será que vou me lembrar de ficar atualizando *totalContas*?



```
1 class ContaBancaria
2 {
3     //outros atributos...
4     private int totalContas;
5
6     public ContaBancaria()
7     {
8         //algum código...
9         totalContas++;
10    }
11    //demais métodos...
12 }
```

- Então basta que a própria classe **ContaBancaria** controle essa informação no seu **construtor**!
- Desse modo, fica garantido que toda vez que um objeto for criado, ele será contabilizado.
- Essa solução funcionará?

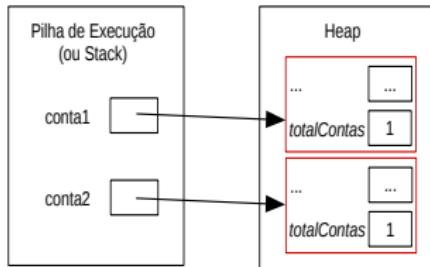
# Atributos Estáticos

- ▶ O que acontece se criarmos dois objetos da classe **ContaBancaria**?

```
ContaBancaria conta1 = new ContaBancaria();  
ContaBancaria conta2 = new ContaBancaria();
```

- ▶ Qual seria o valor do atributo *totalContas*?

- ▶ Como sabemos, cada objeto tem seus valores para seus atributos, ou seja, **o atributo é de cada objeto!**
- ▶ Isso ocorre porque, na memória, cada objeto tem seu espaço para guardar os valores de seus atributos.
- ▶ Portanto, a cada novo objeto da classe **ContaBancaria**, um novo espaço de memória é reservado para armazenar o valor de *totalContas*.



# Atributos Estáticos

E se conseguíssemos reservar um espaço de memória que fosse **compartilhado** por **todos os objetos** de uma mesma classe?



Podemos fazer isso usando um **atributo estático**!

- ▶ Um **atributo estático** pertence à **classe** e não aos objetos.
- ▶ Quando um **atributo** é declarado com **estático**, um espaço de memória (*heap*) é reservado para ele **uma única vez** e pode ser acessado por todos os objetos daquela classe.
- ▶ Desse modo, se um objeto alterar o valor de um atributo estático, todos os demais poderão acessar esse novo valor.

# Atributos Estáticos

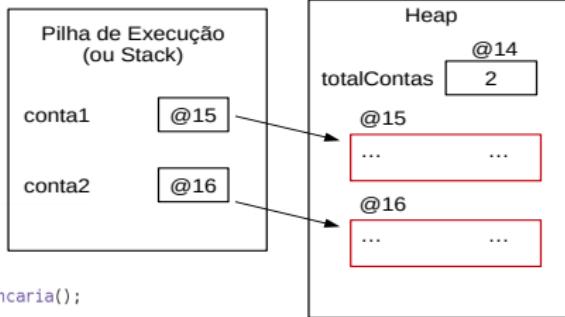
```
1 class ContaBancaria
2 {
3     //outros atributos...
4     private static int totalContas;
5
6     public ContaBancaria()
7     {
8         //algum código...
9         totalContas++;
10    }
11    //demais métodos...
12 }
```

Vamos incluir o atributo estático em nosso código!

A declaração de um atributo **estático** é feita utilizando-se o modificador **static**.



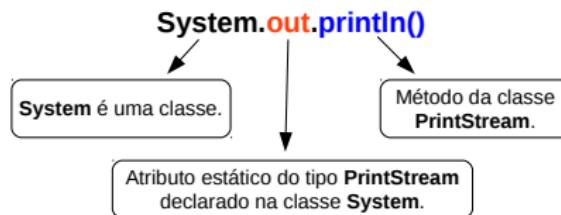
```
1 public class ContaBancariaTeste
2 {
3     public static void main(String[] args)
4     {
5         ContaBancaria conta1 = new ContaBancaria();
6
7         ContaBancaria conta2 = new ContaBancaria(); ←
8     }
9 }
```



Veja que agora temos apenas uma cópia do atributo *totalContas* na memória e, portanto, o incremento dentro do construtor afeta essa cópia. Desse modo, quando um segundo objeto da classe **ContaBancaria** é criado, o atributo *totalContas* que já está com o valor 1 é incrementado e passa a armazenar o valor 2.

# Atributos Estáticos

- ▶ Os atributos estáticos são denominados **variáveis de classe**, o que os diferencia das **variáveis de instância** (que correspondem às implementações dos atributos dos objetos).
- ▶ Um exemplo de atributo estático que já usamos é *out*, um atributo estático público da classe *System*.



- ▶ Todos os atributos estáticos públicos (*public static*) de uma classe podem ser acessados por meio de uma referência a qualquer objeto da classe **ou** qualificando o nome do atributo com o nome da classe e um ponto (.), ou seja, *Classe.atributo*.

# Constantes e Atributos Estáticos

Em linguagens de programação OO não existe o conceito de variável global (acessível de qualquer parte do código).

Então como definir, por exemplo, constantes “globais” nessas linguagens?

- ▶ Em Java, já existem algumas constantes definidas em uma classe contida em um de seus pacotes. Por exemplo, a constante PI é um atributo da classe *Math* definida no pacote *java.lang*.
- ▶ Como acessar o valor de *pi*? Se ele é um atributo de uma classe, basta criar um objeto dessa classe para acessá-lo, correto?
- ▶ Hummm...mas não seria um desperdício de recursos e desempenho ter que criar um objeto da classe *Math* só para consultar seus valores?
- ▶ De fato, não precisamos criar um objeto para isso uma vez que PI é um atributo **estático, público** e constante.
  - ▶ Assim como aprendemos, podemos acessar o valor de *pi* usando: *Math.PI*.

# Constantes em Java

Mas e se eu quiser criar uma constante em um programa Java?

- ▶ Para declarar uma constante em Java você deve usar o modificador *final*.

```
private final int INCREMENTO;
```

- ▶ Atributos constantes podem ser inicializados quando são declarados **OU** nos construtores.
- ▶ Qual a diferença entre essas duas opções de inicialização?
  - ▶ Inicializar constantes em construtores permite que cada objeto da classe tenha um valor diferente para a constante.
  - ▶ Aliás, se a inicialização for na declaração, recomenda-se declará-lo como um atributo estático, evitando-se criar uma cópia separada do atributo para cada objeto da classe.

# Objetos Constantes



Cuidado!

Usar atributos constantes de tipos primitivos (int, float etc.) é tranquilo.

Mas usar **atributos constantes do tipo objeto** exige cautela, pois objetos constantes **não são imutáveis!**

Considere a declaração do seguinte atributo constante:

```
private final ContaBancaria minhaConta = new ContaBancaria();
```

- ▶ Qual o significado dessa declaração?
  - ▶ Significa apenas que não posso atribuir um novo objeto à variável *minhaConta* (seu endereço é constante).
  - ▶ No entanto, nada impede que eu chame os métodos desse objeto e, desse modo, altere seu estado.

# Métodos Estáticos

Em Java, temos um método para calcular o logaritmo natural de um número. Veja como ele é chamado:

```
double valor = Math.log(num);
```

- ▶ Repare que o método *log* está sendo chamado sem que nenhum objeto da classe *Math* tenha sido instanciado.
- ▶ Ele está sendo chamado utilizando-se a sintaxe:  
*Classe.método*.
- ▶ Com o que isso parece?
- ▶ Portanto, assim como os atributos, os **métodos** de uma classe também **podem ser estáticos**.
  - ▶ Nesse caso, eles são chamados de **métodos de classe**.

# Métodos Estáticos

Qual a utilidade de métodos estáticos?

- ▶ Eles são interessantes para realizar operações que não dependem do estado dos objetos.
- ▶ Por isso, eles podem ser executados sem a necessidade de se instanciar um objeto da classe.

# Métodos Estáticos

## Exemplo de Método Estático

- ▶ Suponha que temos uma classe **ContaBancaria**.
- ▶ Essa classe possui um método que converte de R\$ para US\$.  

```
public static double converte(double valorReal, double taxa);
```
- ▶ Você concorda que eu poderia desejar utilizar esse método mesmo sem ter uma conta bancária?
  - ▶ Às vezes precisamos apenas realizar essa conversão e não de uma conta bancária propriamente dita.
- ▶ O que aconteceria se esse método não fosse estático?
  - ▶ O usuário seria obrigado a criar um objeto ContaBancaria só para fazer essa conversão.
  - ▶ Mas não faz sentido criar uma “falsa” conta bancária apenas para poder chamar um método como esse.<sup>1</sup>

---

<sup>1</sup>Em uma boa modelagem esse método provavelmente ficaria em uma classe utilitária (e não em ContaBancaria). No entanto, aqui vale como um exemplo didático.

# Restrições para Métodos Estáticos

```
1 class ContaBancaria
2 {
3     //outros atributos...
4     private double saldo;
5     private static double limiteConta;
6
7     //outros métodos...
8     public static double saldoDisponivel()
9     {
10         return limiteConta + saldo;
11     }
12 }
13 }
```

Existe algum problema com esse método estático?



- ▶ Como o método é estático, ou seja, é um método de classe, ele pode ser chamado sem existir nenhum objeto daquela classe.
- ▶ Portanto, se ele for chamado diretamente – *ContaBancaria.saldoDisponivel()*, qual seria o valor do atributo *saldo*?
- ▶ Ou se existissem vários objetos conta bancária, o saldo a ser usado seria de qual conta?

# Restrições para Métodos Estáticos

```
1 class ContaBancaria
2 {
3     //outros atributos...
4     private double saldo;
5     private double limiteConta;
6
7     //outros métodos...
8     public static double saldoDisponivelDolar(double taxa)
9     {
10         return getSaldo()*taxa;
11     }
12
13     public double getSaldo(){
14         return saldo + limiteConta;
15     }
16 }
```

Este programa  
funciona? Por que?



A mensagem de erro mostra  
que um método não estático  
não pode ser chamado a partir  
de um método estático. Mas  
por qual motivo?

```
luiz@HP:~/TesteProgramas$ javac ContaBancariaTeste.java
./ContaBancaria.java:10: error: non-static method getSaldo()
cannot be referenced from a static context
    return getSaldo()*taxa;
               ^
1 error
```

- ▶ A questão é que um método não estático pode acessar um atributo também não estático.
- ▶ Desse modo, se um método estático pudesse chamar um não estático, poderíamos acessar (indiretamente) atributos não estáticos (o que sabemos não ser possível).

# Restrições para Métodos Estáticos

Lembre-se, quando um método estático é chamado, pode não haver nenhum objeto da sua classe na memória.



Portanto, um método **estático** não pode acessar **atributos não estáticos** nem chamar **métodos não estáticos** da classe<sup>1</sup>.

<sup>1</sup>Exceto se o acesso tiver sendo feito por uma instância da classe criada previamente.

# O Método *main*

- ▶ Agora que já sabemos o que é um método estático, vamos recordar uma declaração que já utilizamos diversas vezes.

public static void main(String[ ] args)

Por que o método *main* precisa ser estático?

- ▶ Para responder essa pergunta vamos lembrar como executamos programas usando a JVM:

**java MinhaClasse**

- ▶ A partir desse comando, o que a JVM faz é realizar a seguinte chamada:

*MinhaClasse.main(...)*<sup>2</sup>

- ▶ Perceba que essa chamada só é possível porque o método *main* é **estático**!

---

<sup>2</sup>Ao executar um programa você pode passar dados para execução do UFLN  
mesmo.

# *Garbage Collector*

Em um programa Java, o que acontece na memória com uma variável local (tipo primitivo) quando um método termina?

- ▶ A variável é sempre desalocada quando o método termina.
- ▶ Como no caso de tipos primitivos a variável armazena seu próprio valor, desalocar uma variável, significa desalocar o seu próprio valor.

Mas o que acontece se essa variável local estiver referenciando um objeto?

- ▶ No caso de variáveis que referenciam objetos, desalocar a variável significa que o local que armazena o endereço do objeto foi desalocado.
  - ▶ Mas e a memória alocada para o objeto (*heap*) em si, o que acontece?

# *Garbage Collector*



É aí que entra em ação o *Garbage Collector* (coletor de lixo)!

Sempre que um objeto já **não é mais acessível**, ou seja, quando perdemos a referência para esse objeto:

- ▶ Ele fica disponível para o coletor de lixo (*garbage collector*).
- ▶ Mas atenção, o *Garbage Collector* age apenas sobre objetos, nunca sobre variáveis.
- ▶ Portanto, você não precisa se preocupar em desalocar a memória utilizada por um objeto, pois Java cuida disso para você.

# *Garbage Collector*

Vamos analisar a situação a seguir:

Até este momento temos 2 objetos em memória.



```
public class ContaBancariaTeste
{
    public static void main(String[] args)
    {
        ContaBancaria conta1 = new ContaBancaria();
        ContaBancaria conta2 = new ContaBancaria();
    }
}
```

Agora vamos executar a seguinte linha de código:

```
conta2 = conta1
```

Quantos objetos temos em memória **agora?**

Apenas um? **Não podemos afirmar isso!** O *Garbage Collector* é controlado pela JVM. Portanto, não tem como saber quando exatamente um objeto será desalocado.

Executamos uma linha de código que nos fez perder a referência para um dos objetos criados.

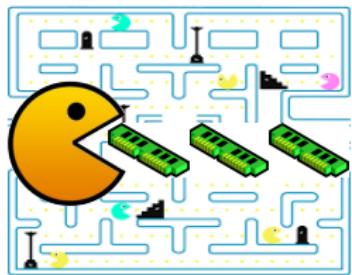


# Vazamento de Memória



O vazamento de memória ocorre quando **deixamos** em nosso programa objetos que **não estão mais sendo usados alocados na memória**, fazendo com que o coletor de lixo não consiga reconhecê-los como objetos elegíveis para desalocação.

Como resultado, estes objetos **nunca são desalocados**, permanecem na memória indefinidamente, **reduzindo** assim a quantidade total de **memória disponível**. Num sistema que precise funcionar continuamente (nunca é desligado), isso levará a um travamento do sistema ou do computador.



# Vazamento de Memória

Uma vez que o Java possui o *Garbage Collector*, precisamos nos preocupar com vazamento de memória?

- ▶ O Gabage Collector **ajuda a evitar** vazamento de memória, uma vez que ele desaloca memória ocupada por objetos que **não** estão mais sendo **referenciados**.
- ▶ No entanto, ele **não elimina inteiramente** os vazamentos de memória! Por que?
  - ▶ Ele **não** efetuará **coleta** de lixo de um objeto até que **não** haja nenhuma **referência** a ele.
  - ▶ Portanto, se você mantiver **erroneamente** referências a objetos indesejáveis, **vazamentos de memória podem ocorrer**.

# Destruidores

Java não possui destrutor!

- ▶ Com o *garbage collector*, vazamentos de memória que são comuns em outras linguagens (tais como em *C* e *C++*, já que a memória não é automaticamente desalocada), são menos prováveis em *Java*.
- ▶ No entanto, além de vazamento de memória, **vazamentos de outros recursos** podem ocorrer.



# Destruidores

## Vazamento de outros recursos

- ▶ Por exemplo, um aplicativo pode abrir um arquivo no disco para modificar seu conteúdo.
- ▶ Se o aplicativo não fechar o arquivo depois que ele não é mais necessário, pode acontecer de ele não estar disponível para uso em outros programas.

Portanto, é importante que:

- ▶ Liberemos um recurso, como um arquivo ou uma conexão com um banco de dados, depois que eles não forem mais necessários.
- ▶ Para isso, podemos utilizar uma cláusula chamada *finally*<sup>1</sup> indicando o que deve ser feito para liberar um recurso.
- ▶ Desse modo, teremos a certeza de que aquele recurso (arquivo ou conexão) será liberado, ainda que algo tenha falhado no decorrer do código.

---

<sup>1</sup>Há também, no Java 7, um recurso conhecido como *try-with-resources* da UFSC  
que permite utilizar a semântica do *finally* de maneira mais simples.

# Perguntas?

